Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation

Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H J Kelly

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2BZ, United Kingdom
{o.beckmann,p.kelly}@imperial.ac.uk
www.doc.ic.ac.uk/{~ob3,~phjk}

Abstract. The TaskGraph Library is a C++ library for dynamic code generation, which combines specialisation with dependence analysis and loop restructuring. A TaskGraph represents a fragment of code which is constructed and manipulated at runtime, then compiled, dynamically linked and executed. TaskGraphs are initialised using macros and overloading, which forms a simplified, C-like sub-language with first-class arrays and no pointers. Once a TaskGraph has been constructed, we can analyse its dependence structure and perform optimisations. In this Chapter, we present the design of the TaskGraph library, and two sample applications to demonstrate its use for runtime code specialisation and restructuring optimisation.

1 Introduction

The work we describe in this Chapter is part of a wider research programme at Imperial College aimed at addressing the apparent conflict between the quality of scientific software and its performance. The TaskGraph library, which is the focus of this Chapter, is a key tool which we are developing in order to drive this research programme. The library is written in C++ and is designed to support a model of software components which can be composed dynamically, and optimised, at runtime, to exploit execution context:

– *Optimisation with Respect to Runtime Parameters*
  The TaskGraph library can be used for specialising software components according to either their parameters or other runtime context information. Later in this Chapter (Sec. 3), we show an example of specialising a generic image filtering function to the particular convolution matrix being used.
– *Optimisation with Respect to Platform*
  The TaskGraph library uses SUIF-1 [1], the Stanford University Intermediate Format, as its internal representation for code. This makes a rich collection of dependence analysis and restructuring passes available for our use in code optimisation. In Sec. 5 of this Chapter we show an example of generating, at runtime, a matrix multiply component which is optimally tiled with respect to the host platform.

The TaskGraph library is a tool for implementing domain-specific optimisations in active libraries for high-performance computing; we discuss its relationship with the other work in this book in Sec. 8, which concludes the Chapter.

Background. Several earlier tools for dynamic code optimisation have been reported in the literature [2, 3]. The key characteristics which distinguish our approach are as follows:

- *Single-Language Design*
  The TaskGraph library is implemented in C++ and any TaskGraph program can be compiled as C++ using widely-available compilers. This is in contrast with approaches such as Tick-C [2] which rely on a special compiler for processing dynamic constructs. The TaskGraph library's support for manipulating code as data within one language was pioneered in Lisp [4].
- *Explicit Specification of Dynamic Code*
  Like Tick-C [2], the TaskGraph library is an imperative system in which the application programmer has to construct the code as an explicit data structure. This is in contrast with ambitious partial evaluation approaches such as DyC [3, 5] which use declarative annotations of regular code to specify where specialisation should occur and which variables can be assumed constant. Offline partial evaluation systems like these rely on *binding-time analysis* (BTA) to find other, derived static variables [6].
- *Simplified C-like Sub-language*
  Dynamic code is specified with the TaskGraph library via a small sub-language which is very similar to standard C (see Sec. 2). This language has been implemented through extensive use of macros and C++ operator overloading and consists of a small number of special control flow constructs, as well as special types for dynamically bound variables. Binding times of variables are explicitly determined by their C++ types, while binding times for intermediate expression values are derived using C++'s overloading mechanism (Sec. 4). The language has first-class arrays, unlike C and C++, to facilitate dependence analysis.

In Sec. 6 we discuss the relationship with other approaches in more detail.

Structure of this Chapter. In Sec. 2, we describe how TaskGraphs are constructed. Section 3 offers a simple demonstration of runtime specialisation. Section 4 explains how the library itself is implemented. In Sec. 5, we use matrix multiplication to illustrate the use of the library's loop restructuring capabilities. In Secs. 6 and 7 we discuss related and ongoing work, and Sec. 8 concludes.

2 The TaskGraph Library API

A TaskGraph is a data structure which holds the abstract syntax tree (AST) for a piece of dynamic code. A key feature of our approach is that the application programmer has access to and can manipulate this data structure at

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <TaskGraph>
4  using namespace tg;
5  int main( int argc, char *argv[] ) {
6    TaskGraph T;
7    int b = 1;
8    int c = atoi( argv [1] );
9    taskgraph( T ) {
10     tParameter( tVar( int, a ) );
11     a = a + c;
12   }
13   T.compile();
14   T.execute( "a", &b, NULL );
15   printf ( "b = %d\n", b );
16 }
```

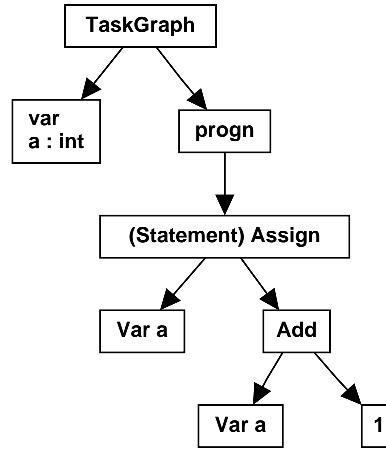

Fig. 1. *Left:* Simple Example of using the TaskGraph library. *Right:* Abstract syntax tree (AST) for the simple TaskGraph constructed by the piece of code shown on the left. The `int` variable `c`, is static at TaskGraph construction time, and appears in the AST as a value (see Sec. 4). The (not type-safe) `execute()` call takes a NULL-terminated list of parameter name/value pairs; it binds TaskGraph parameter "`a`" to the address of the integer `b`, then invokes the compiled code.

runtime; in particular, we provide an extensible API (sub-language) for *constructing* TaskGraphs at runtime. This API was carefully designed using macros and C++ operator overloading to look as much as possible like ordinary C.

A Simple Example. The simple C++ program shown in the left-hand part of Fig. 1 is a complete example of using the TaskGraph library. When compiled with `g++`, linked against the TaskGraph library and executed, this program dynamically creates a piece of code for the statement `a = a + c`, binds the application program variable `b` as a parameter and executes the code, printing `b = 2` as the result. This very simple example illustrates both that creation of dynamic code is completely explicit in our approach and that the language for creating the AST which a TaskGraph holds looks similar to ordinary C.

3 Generalised Image Filtering

We now show an example which uses a fuller range of TaskGraph constructs and which also demonstrates a real performance benefit from runtime code optimisation. A generic image convolution function, which allows the application programmer to supply an arbitrary convolution matrix could be written in C as shown in Fig. 2. This function has the advantage of genericity (the interface is in principle similar to the General Linear Filter functions from the Intel Performance Libraries [7, Sec. 9]) but suffers from poor performance because

```
void convolution( const int IMGSZ, const float *image, float *new_image,
                  const int CSZ /* convolution matrix size */, const float *matrix ) {
  int i, j, ci, cj; const int c_half = ( CSZ / 2 );

  // Loop iterating over image
  for( i = c_half; i < IMGSZ - c_half; ++i ) {
    for( j = c_half; j < IMGSZ - c_half; ++j ) {
      new_image[i * IMGSZ + j] = 0.0;

      // Loop to apply convolution matrix
      for( ci = - c_half; ci <= c_half; ++ci ) {
        for( cj = - c_half; cj <= c_half; ++cj ) {
          new_image[i * IMGSZ + j] += ( image[(i+ci) * IMGSZ + j+cj] *
                                        matrix[(c_half+ci) * CSZ + c_half+cj] );
} } } } }
```

Fig. 2. Generic image filtering: C++ code. Because the size as well as the entries of the convolution matrix are runtime parameters, the inner loops (for-ci and for-cj), with typically very low trip-count, cannot be unrolled efficiently.

- The bounds of the inner loops over the convolution matrix are statically unknown, hence these loops, with typically very low trip-count, cannot be unrolled efficiently.
- Failure to unroll the inner loops leads to unnecessarily complicated control flow and also blocks optimisations such as vectorisation on the outer loops.

Figure 3 shows a function which constructs a TaskGraph that is specialised to the particular convolution matrix being used. The `tFor` constructs are part of the TaskGraph API and create a loop node in the AST. Note, however, that the inner `for` loops are executed as ordinary C++ at TaskGraph *construction* time, creating an assignment node in the AST for each iteration of the loop body. The effect is that the AST contains control flow nodes for the for-i and for-j loops and a loop body consisting of `CSZ * CSZ` assignment statements.

We study the performance of this example in Fig. 4. The convolution matrix used was a 3×3 averaging filter, images were square arrays of single-precision floats ranging in size up to 4094×4096. Measurements are taken on a Pentium 4-M with 512KB L2 cache running Linux 2.4, `gcc` 3.3 and the Intel C++ compiler version 7.1. We compare the performance of the following:

- The static C++ code, compiled with `gcc` 3.3 (-O3).
- The static C++ code, compiled with the Intel C++ compiler version 7.1 (-restrict -O3 -ipo -xiMKW -tpp7 -fno-alias). The `icc` compiler reports that the innermost loop (for-cj) has been vectorised[1]. Note, however, that this loop will have a dynamically determined trip-count of 3, i.e. the Pentium 4's 16-byte vector registers will not be filled.
- The code dynamically generated by the TaskGraph library, compiled with `gcc` 3.3 (-O3). The two innermost loops are unrolled.

[1] The SSE2 extensions implemented on Pentium 4 processors include 16-byte vector registers and corresponding instructions which operate simultaneously on multiple operands packed into them [8].

```
1  void taskgraph_convolution( TaskGraph &T, const int IMGSZ,
2                              const int CSZ, const float *matrix ) {
3    int ci, cj;
4    assert ( CSZ % 2 == 1 );
5    const int c_half = ( CSZ / 2 );
6
7    taskgraph( T ) {
8      unsigned int dims[] = {IMGSZ * IMGSZ};
9      tParameter( tArrayFromList( float, tgimg, 1, dims ) );
10     tParameter( tArrayFromList( float, new_tgimg, 1, dims ) );
11     tVar ( int, i );
12     tVar ( int, j );
13
14     // Loop iterating over image
15     tFor( i, c_half, IMGSZ − (c_half + 1) ) {
16       tFor( j, c_half, IMGSZ − (c_half + 1) ) {
17         new_tgimg[i * IMGSZ + j] = 0.0;
18
19         // Loop to apply convolution matrix
20         for( ci = −c_half; ci <= c_half; ++ci ) {
21           for( cj = −c_half; cj <= c_half; ++cj) {
22             new_tgimg[i * IMGSZ + j] +=
23               tgimg[(i+ci) * IMGSZ + j+cj] * matrix[(c_half+ci) * CSZ + c_half+cj];
24           } } } }
25   }
26 }
```

Fig. 3. Generic image filtering: function constructing the TaskGraph for a specific convolution matrix. The size as well as the entries of the convolution matrix are static at TaskGraph construction time. This facilitates complete unrolling of the inner two loops. The outer loops (for-i and for-j) are entered as control flow nodes in the AST.

- The code dynamically generated by the TaskGraph library, compiled with `icc` 7.1 (-restrict -O3 -ipo -xiMKW -tpp7 -fno-alias). The two innermost loops are unrolled and the then-remaining innermost loop (the for-j loop over the image) is vectorised by `icc`.

In a realistic image filtering application, the datatype would probably be fixed-precision integers, or, in more general scientific convolution applications, double-precision floats. We chose single-precision floats here in order illustrate the fact that specialisation with respect to the bounds of the inner loop can facilitate efficient vectorisation.

We have deliberately measured the performance of these image filtering functions for only one pass over an image. In order to see a real speedup the overhead of runtime compilation therefore needs to be recovered in just a single application of the generated code. Figure 4 shows that we do indeed get an overall speedup for image sizes that are greater than 1024×1024. In the right-hand part of Fig. 4, we show a breakdown of the overall execution time for two specific data sizes. This demonstrates that although we achieve a huge reduction in execution time of the actual image filtering code, the constant overhead of runtime compilation cancels out this benefit for a data size of 1024×1024. However, for larger data sizes, we achieve an overall speedup.

Note, also, that image filters such as the one in this example might be applied either more than once to the same image or to different images — in either case,

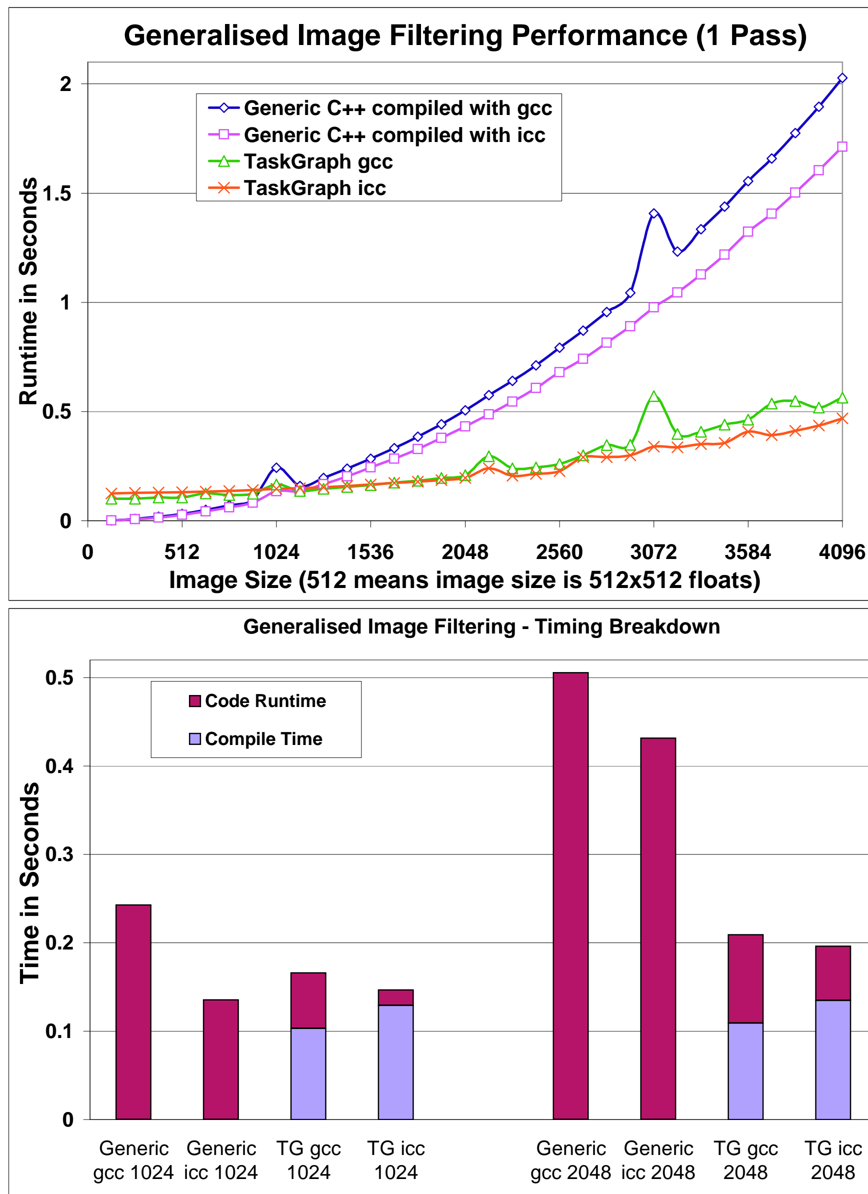

Fig. 4. Performance of image filtering example. Top: Total execution time, including runtime compilation, for one pass over image. Bottom: Breakdown of total execution time into compilation time and execution time of the actual convolution code for two specific image sizes: 1024×1024 (the break-even point) and 2048×2048.

we would have to pay the runtime compilation overhead only once and will get higher overall speedups.

4 How it Works

Thus far, we have given examples of how the TaskGraph library is used, and demonstrated that it can achieve significant performance gains. In this section we now give a brief overview of TaskGraph syntax, together with an explanation of how the library works.

TaskGraph Creation. The TaskGraph library can represent code as data — specifically, it provides TaskGraphs as data structures holding the AST for a piece of code. We can create, compile and execute different TaskGraphs independently. Statements such as the assignment `a = a + c` in line 11 of Fig. 1 make use of C++ operator overloading to add nodes (in this case an assignment statement) to a TaskGraph. Figure 1 illustrates this by showing a graphical representation of the complete AST which was created by the adjacent code. Note that the variable `c` has *static* binding-time for this TaskGraph. Consequently, the AST contains its value rather than a variable reference.

The `taskgraph( T ){...}` construct (see line 7 in Fig. 3) determines which AST the statements in a block are attached to. This is necessary in order to facilitate independent construction of different TaskGraphs.

Variables in TaskGraphs. The TaskGraph library inherits lexical scoping from C++. The `tVar(type, name)` construct (see lines 11 and 12 in Fig. 3) can be used to declare a dynamic local variable.

Similarly, the `tArray(type, name, no_dims, extents[])` construct can be used to declare a dynamic multi-dimensional array with number of dimensions `no_dims` and size in each dimension contained in the integer array `extents`. Arrays are first-class objects in the TaskGraph construction sub-language and can only be accessed inside a TaskGraph using the `[]` subscript operators. There are no pointers in the TaskGraph construction sub-language.

TaskGraph Parameters. Both Fig. 1 (line 10) and Fig. 3 (lines 9 and 10) illustrate that any TaskGraph variable can be declared to be a TaskGraph parameter using the `tParameter()` construct. We require the application programmer to ensure that TaskGraph parameters bound at execution time do not alias each other.

Control Flow Nodes. Inside a TaskGraph construction block, `for` loops and `if` conditionals are executed at construction time. Therefore, the `for` loops on lines 20 and 21 in Fig. 3 result in an unrolled inner loop. However, the TaskGraph sub-language defines some constructs for adding control-flow nodes to an AST: `tFor(var,lower,upper)` adds a loop node (see lines 15 and 16 in Fig. 3). The loop bounds are *inclusive*. `tIf()` can be used to add a conditional node to the AST. The TaskGraph embedded sublanguage also includes `tWhile`, `tBreak` and `tContinue`. Function call nodes can be built, representing execution-time calls to functions defined in the host C++ program.

```
303   taskgraph( T ) {
304     unsigned int dims[] = {IMGSZ * IMGSZ};
305     tParameter( tArrayFromList( float, tgimg , 1, dims ) );
306     tParameter( tArrayFromList( float, new_tgimg , 1, dims ) );
307     tVar ( int, i );
308     tVar ( int, j );
309
310     // Loop iterating over image
311     tFor( i , c_half, IMGSZ − (c_half + 1) ) {
312       tFor( j , c_half, IMGSZ − (c_half + 1) ) {
313         new_tgimg [ i * IMGSZ + j ] = 0.0;
314
315         // Loop to apply convolution matrix
316         for( ci = −c_half; ci <= c_half; ++ci ) {
317           for( cj = −c_half; cj <= c_half; ++cj) {
318             new_tgimg [ i * IMGSZ + j ] +=
319               tgimg [( i +ci) * IMGSZ + j +cj] * matrix[(c_half+ci) * CSZ + c_half+cj];
320           } } } }
321   }
322 }
```

Fig. 5. Binding-Time Derivation. TaskGraph construction code for the image filtering example from Fig. 2, with all dynamic variables marked by a boxed outline.

Expressions and Binding-Time. We refer to variables that are bound at TaskGraph construction time as *static* variables and those that are bound at execution time as *dynamic*. Declarative code specialisation systems such as DyC [3] use annotations that declare some variables to be static for the purpose of partial evaluation. In contrast, static binding time, i.e. evaluated at TaskGraph construction time is the default for the TaskGraph language. Only TaskGraph variables and parameters are dynamic; this is indicated explicitly by declaring them appropriately (`tVar`, `tParameter`). The overloaded operators defined on those dynamic types define binding-time derivation rules. Thus, an expression such as `a + c` in Fig. 1 where `a` is dynamic and `c` is static is derived dynamic, but the static part is evaluated at construction time and entered into the AST as a value. We illustrate this by reproducing the TaskGraph image filtering code from Fig. 3 again in Fig. 5; however, this time all dynamic expressions are marked by a boxed outline. Note that the convolution matrix, including its entire subscript expression in the statement on line 22, is *static*.

5 Another Example: Matrix Multiply

In Sec. 3, we showed an example of how the specialisation functionality of the TaskGraph library can be used to facilitate code optimisations such as vectorisation. In this Section, we show, using matrix multiplication as an example, how we can take advantage of the use of SUIF-1 as the underlying code representation in the TaskGraph library to perform restructuring optimisations at runtime.

```
/*
 * mm_ijk
 * Most straight−forward matrix multiply
 * Calculates C += A * B
 */
void mm_ijk( const unsigned int sz,
             const float *const A,
             const float *const B,
             float *const C ) {
  unsigned int i, j, k;
  for( i = 0; i < sz; ++i ) {
    for( j = 0; j < sz; ++j ) {
      for( k = 0; k < sz; ++k ) {
        C[i*sz+j] += A[i*sz+k] * B[k*sz+j];
      }
    }
  }
}
```

```
void TG_mm_ijk( unsigned int sz[2],
                TaskLoopIdentifier *loop,
                TaskGraph &t ) {
  taskgraph( t ) {
    tParameter(tArrayFromList(float,A,2,sz));
    tParameter(tArrayFromList(float,B,2,sz));
    tParameter(tArrayFromList(float,C,2,sz));
    tVar(int,i); tVar(int,j); tVar(int,k);

    tGetId( loop [0] );
    tFor( i , 0, sz [0] − 1 ) {
      tGetId( loop [1] );
      tFor( j , 0, sz [1] − 1 ) {
        tGetId( loop [2] );
        tFor( k , 0, sz [0] − 1 ) {
          C[i ][ j] += A[i][k] * B[k][j ];

} } } } }
```

```
for( int tsz = 4; tsz <= min(362, matsz); ++tsz ) {
  for( int i = 0; i < samples; ++i ) {
    unsigned int sizes[] = { matsz, matsz };
    int trip3 [] = { tsz , tsz , tsz };
    TaskLoopIdentifier loop [3];
    TaskGraph MM;

    TG_mm_ijk( sizes, loop, MM );
    interchangeLoops( loop [1], loop [2] ); // Interchange loops
    tileLoop( 3, &loop [0], trip3 );        // Tile inner two loops
    MM.compile( TaskGraph::ICC, false );

    tt3 = time_function ();
    MM.setParameters( "A", A, "B", B, "C", C, NULL );
    MM.execute();
    tt3 = time_function() − tt3;

    time[i ]  = time_to_seconds( tt3 );
  }
  time[samples]  = mean( samples, time );
  if( time[samples] < best_time_gcc ) {
    best_time_gcc = time[samples];
    best_tsz_gcc  = tsz;
  }
}
```

Fig. 6. The code on the top left is the standard C++ matrix multiply (ijk loop order) code. The code on the top right constructs a TaskGraph for the standard ijk matrix multiply loop. The code underneath shows an example of using the TaskGraph representation for the ijk matrix multiply kernel, together with SUIF-1 passes for interchanging and tiling loops to search for the optimal tilesize of the interchanged and tiled kernel for a particular architecture and problem size.

Figure 6 shows both the code for the standard C/C++ matrix multiply loop (ijk loop order) and the code for constructing a TaskGraph representing this loop, together with an example of how we can direct optimisations from the application program: we can interchange the for-j and for-k loops before compiling and executing the code. Further, we can perform loop tiling with a runtime-selected tile size. This last application demonstrates in particular the possibilities of using the TaskGraph library for domain-specific optimisation:

- *Optimising for a particular architecture*
  In Fig. 6, we show a simple piece of code which implements a runtime search for the optimal tilesize when tiling matrix multiply. In Fig. 8, we show the results of this search for both a Pentium 4-M (with 512K L2 cache) and an Athlon (with 256K L2 cache) processor. The resulting optimal tilesizes differ for most problem sizes, but they do not differ by as much as would have been expected if the optimal tilesize was based on L2 capacity. We assume that a different parameter, such as TLB (translation lookaside buffer) span, is more significant in practice.
- *Optimising for a particular loop or working set*
  The optimal tile size for matrix multiply calculated by our code shown in Fig. 6 differs across problem sizes (see Fig. 8). Similarly, we expect the optimal tilesize to vary for different loop bodies and resulting working sets.

We believe that performance achieved with relatively straight-forward code in our matrix multiply example (up to 2 GFLOP/s on a Pentium 4-M 1.8 GHz, as shown in Fig. 7) illustrates the potential for the approach, but to achieve higher performance we would need to add hierarchical tiling, data copying to avoid associativity conflicts, unrolling and software pipelining; the ATLAS library [9], for comparison, achieves more than 4 GFLOPS on the Pentium 4-M above. Our approach allows applying at least some of the optimisations which are used by the ATLAS library to achieve its very high performance to TaskGraphs that have been written by the application programmer.

6 Related Work

In this section, we briefly discuss related work in the field of dynamic and multi-stage code optimisation.

Language-Based Approaches.

- *Imperative*
  Tick-C or 'C [2], a superset of ANSI C, is a language for dynamic code generation. Like the TaskGraph library, 'C is explicit and imperative in nature; however, a key difference in the underlying design is that 'C relies on a special compiler (`tcc`). Dynamic code can be specified, composed and instantiated, i.e. compiled, at runtime. The fact that 'C relies on a special compiler also means that it is in some ways a more expressive and more powerful system

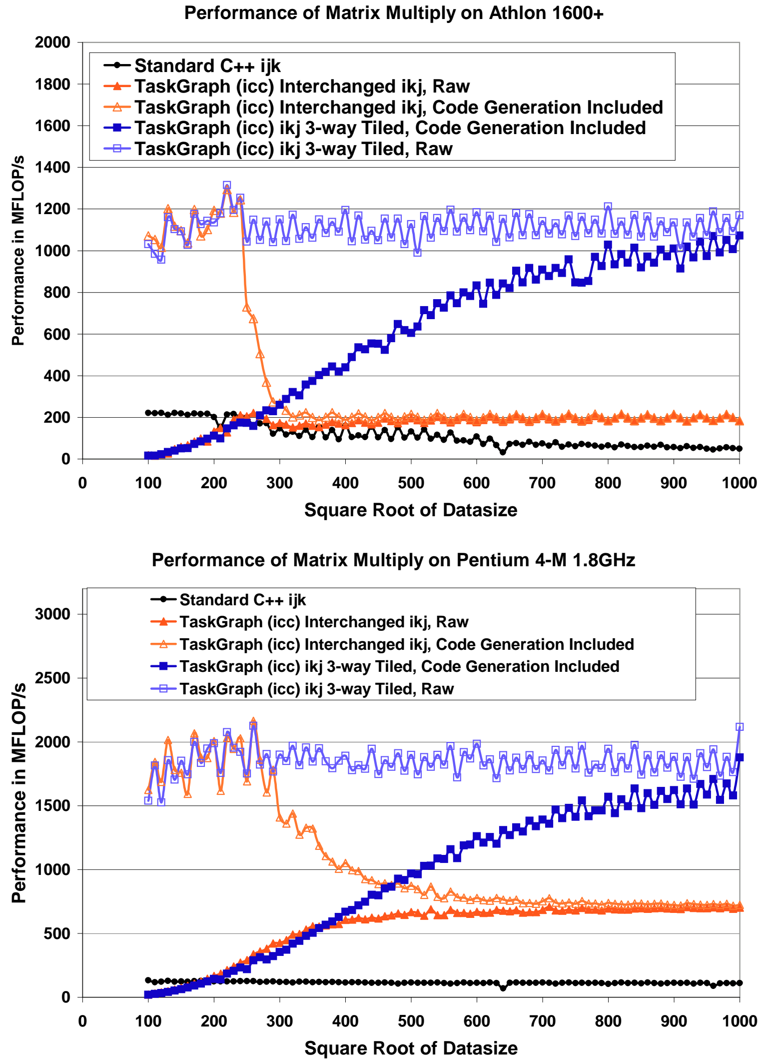

Fig. 7. Performance of single-precision matrix multiply on Athlon 1600+ with 256KB L2 cache and on Pentium 4-M 1.8 GHz with 512KB L2 cache. We show the performance of the naive C++ code (ijk loop order, compiled with `gcc` 3.3, -O3), the code where we have used the TaskGraph library to interchange the inner two loops (resulting in ikj loop order) and the code where the TaskGraph library is used to interchange and 3-way tile the loops. For the tiled code, we used the TaskGraph library to search for the optimal tile size for each data point, as shown in Fig. 6. For both the interchanged and tiled code, we plot one graph showing the raw performance of the generated code and one graph which shows the performance after the dynamic code generation cost has been amortised over one invocation of the generated code. All TaskGraph generated code was compiled with `icc` 7.1, -restrict -O3 -ipo -xiMKW -tpp7 -fno-alias.

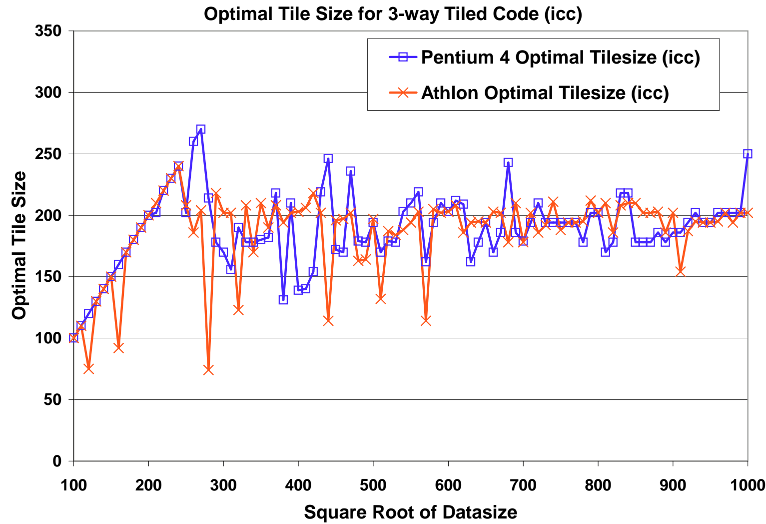

Fig. 8. Optimal tile size on Athlon and Pentium 4-M processors, for each data point from Fig. 7. These results are based on a straight-forward exhaustive search implemented using the TaskGraph library's runtime code restructuring capabilities (see code in Fig. 6).

than the TaskGraph library. For example, 'C facilitates the construction of dynamic function calls where the type and number of parameters is dynamically determined. This is not possible in the TaskGraph library. Jak [10], MetaML [11], MetaOCaml [12] and Template Haskell [13] are similar efforts, all relying on changes to the host language's syntax. Some of what we do with the TaskGraph library can be done using template metaprogramming in C++, which is used, for example, for loop fusion and temporary elimination in the Blitz++ and POOMA libraries [14, 15].

– *Declarative*
DyC [3,5] is a dynamic compilation system which specialised selected parts of programs at runtime based on runtime information, such as values of certain data structures. DyC relies on declarative user annotations to trigger specialisation. This means that a sophisticated binding-time analysis is required which is both polyvariant (i.e. allowing specialisation of one piece of code for different combinations of static and dynamic variables) and program-point specific (i.e. allowing polyvariant specialisation to occur at arbitrary program points). The result of BTA is a set of *derived* static variables in addition to those variables which have been annotated as static. In order to reduce runtime compilation time, DyC produces, at compile-time, a *generating extension* [6] for each specialisation point. This is effectively a dedicated compiler which has been specialised to compile only the code which is being

dynamically optimised. This static pre-planning of dynamic optimisation is referred as *staging*.
Marlet et al. [16] present a proposal for making the specialisation process itself more efficient. This is built using Tempo [17], an offline partial evaluator for C programs and also relies on an earlier proposal by Glück and Jørgensen to extend two-level binding-time analysis to multiple levels [18], i.e. to distinguish not just between dynamic and static variables but between multiple stages. The main contribution of Marlet et al. is to show that multi-level specialisation can be achieved more efficiently by repeated, incremental application of a two-level specialiser.

Data-Flow Analysis. Our library performs runtime data flow analysis on loops operating on arrays. A possible drawback with this solution could be high runtime overheads. Sharma et al. present deferred data-flow analysis (DDFA) [19] as a possible way of combining compile-time information with only limited runtime analysis in order to get accurate results. This technique relies on comprising the data flow information from *regions* of the control-flow graph into *summary functions*, together with a runtime *stitcher* which selects the applicable summary function, as well as computes summary function compositions at runtime.

Transparent Dynamic Optimisation of Binaries. One category of work on dynamic optimisation which contrasts with ours are approaches which do not rely on program source code but instead work in a transparent manner on running binaries. Dynamo [20] is a transparent dynamic optimisation system, implemented purely in software, which works on an executing stream of native instructions. Dynamo interprets the instruction stream until a *hot trace* of instructions is identified. This is then optimised, placed into a code cache and executed when the starting-point is re-encountered. These techniques also perform runtime code optimisation; however, as stated in Sec. 1, our objective is different: restructuring optimisation of software components with respect to context at runtime.

7 Ongoing and Future Work

We have recently evaluated the current TaskGraph library implementation in the context of some moderately large research projects [21]. This experience has led us to planning future developments of this work.

- *Automatic Generation of OpenMP Annotations*
  We would like to use the runtime dependence information which is calculated by the TaskGraph library for automatically annotating the generated code with OpenMP [22] directives for SMP parallelisation. An alternative approach would be to use a compiler for compiling the generated code that has built-in SMP parallelisation capabilities.
- *Automatic Derivation of Component Metadata*
  Our delayed evaluation, self-optimising (DESO) library of data-parallel numerical routines [23] currently relies on hand-written metadata which char-

acterise the data placement constraints of components to perform cross-component data placement optimisation. One of the outstanding challenges which we would like to address in this work is to allow application programmers to write their own data-parallel components *without* having to understand and supply the placement-constraint metadata. We hope to generate these metadata automatically with the help of the TaskGraph library's dependence information. Some initial work on this project has been done [24].

– *Transparent Cross-Component Loop Fusion*
  In an ongoing project [21] we are using the TaskGraph library to perform cross-component loop fusion in our DESO library of data-parallel numerical routines. This works by appending the taskgraphs representing each routine, then applying dependence analysis on adjacent loops to determine when fusion would be valid.

8 Conclusions and Discussion

We present the TaskGraph library as a tool for developing domain-specific optimisations in high-performance computing applications. The TaskGraph sublanguage can be used in two distinct ways:

1. as an embedded language for constructing domain-specific components, which may be specialised to runtime requirements or data, and
2. as a component composition language, supporting applications which explicitly construct, optimise then execute composite computational plans.

In this volume, Lengauer [25] characterises four approaches to delivering domain-specific optimisations. Our goal is to offer a tool for building active libraries, which exploits his "two compilers" approach within a uniform and flexible framework.

The library illustrates the potential for embedding a domain-specific language in C++. The same technique could be used to embed languages for other domains, but we focused on classical scientific applications involving loops over arrays – in order to exploit the powerful techniques of restructuring and parallelising compiler research.

It is common to use template meta-programming in C++ to do this [26]. As surveyed by Czarnecki et al. in this volume [27], it is very attractive to design a language to support program generation as a first-class feature — as do, for example, Template Haskell and MetaOCaml.

We chose a dynamic, runtime approach (like MetaOCaml) to support our long-term goals of re-optimising software to adapt to changing context. This context may be the underlying CPU – as illustrated in Fig. 8, where we search the space of available transformations to best exploit the processor and memory system. We also aim to adapt to the shape and structure of changing data structures, such as a sparse matrix or an adaptive mesh. Similarly with resources, such as the number of processors or network contention. Finally, we're interested

in dynamic composition of components, for example in visualisation and data analysis.

The next step with this research – apart from applying it and encouraging others to do so – is to explore how to build domain-specific optimisations. The challenge here is to make building new optimisations easy enough for domain specialists. We need to build on (or replace) SUIF to support a rewriting-based approach to optimisation, as explored by Visser in this volume [28], and to facilitate extensions to the intermediate representation to support domain-specific program analyses. Of course, the real goal is to use such tools to extend our understanding of program optimisation.

Acknowledgements. This work was supported by the United Kingdom EPSRC-funded OSCAR project (GR/R21486). We thank the referees for helpful and interesting comments.

References

1. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: SUIF: an infrastructure for research on parallelizing and optimizing compilers. ACM SIGPLAN Notices **29** (1994) 31–37
2. Engler, D.R., Hsieh, W.C., Kaashoek, M.F.: 'C: a language for high-level, efficient, and machine-independent dynamic code generation. In: POPL '96: Principles of Programming Languages. (1996) 131–144
3. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: An expressive annotation-directed dynamic compiler for C. Theoretical Computer Science **248** (2000) 147–199
4. McCarthy, J.: History of LISP. In: The first ACM SIGPLAN Conference on History of Programming Languages. Volume 13(8) of ACM SIGPLAN Notices. (1978) 217–223
5. Grant, B., Philipose, M., Mock, M., Chambers, C., Eggers, S.J.: An evaluation of staged run-time optimizations in DyC. In: PLDI '99: Programming Language Design and Implementation. (1999) 293–304
6. Jones, N.D.: Mix Ten Years Later. In: PEPM '95: Partial Evaluation and Semantics-Based Program Manipulation. (1995)
7. Intel Corporation: Integrated Performance Primitives for Intel Architecture. Reference Manual. Volume 2: Image and Video Processing. (200–2001)
8. Intel Corporation: Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual. (1999–2002) Available via `developer.intel.com`.
9. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing **27** (2001) 3–35
10. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: Fifth International Conference on Software Reuse, IEEE Computer Society Press (1998) 143–153
11. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science **248** (2000) 211–242
12. Taha, W.: A gentle introduction to multi-stage programming (2004) In this volume.

13. Sheard, T., Peyton-Jones, S.: Template meta-programming for Haskell. ACM SIGPLAN Notices **37** (2002) 60–75
14. Veldhuizen, T.L.: Arrays in Blitz++. In: ISCOPE'98: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments. Number 1505 in LNCS, Springer-Verlag (1998) 223ff
15. Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W.J., Reynders, J., Smith, S., Williams, T.: Array design and expression evaluation in POOMA II. In: ISCOPE'98: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments. Number 231–238 in LNCS (1998) 223 ff
16. Marlet, R., Consel, C., Boinot, P.: Efficient incremental run-time specialization for free. ACM SIGPLAN Notices **34** (1999) 281–292 Proceedings of PLDI'99.
17. Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.N.: Tempo: Specializing systems applications and beyond. ACM Computing Surveys **30** (1998)
18. Glück, R., Jørgensen, J.: Fast binding-time analysis for multi-level specialization. In: Perspectives of System Informatics. Number 1181 in LNCS (1996) 261–272
19. Sharma, S., Acharya, A., Saltz, J.: Deferred Data-Flow Analysis. Technical Report TRCS98-38, University of California, Santa Barbara (1998)
20. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: A transparent dynamic optimization system. In: PLDI '00: Programming Language Design and Implementation. (2000) 1–12
21. Fordham, P.: Transparent run-time cross-component loop fusion. MEng Thesis, Department of Computing, Imperial College London (2002)
22. `www.opnemp.org`: OpenMP C and C++ Application Program Interface, Version 2.0 (2002)
23. Liniker, P., Beckmann, O., Kelly, P.H.J.: Delayed evaluation self-optimising software components as a programming model. In: Euro-Par 2002: Proceedings of the 8th International Euro-Par Conference. Number 2400 in LNCS (2002) 666–673
24. Subramanian, M.: A C++ library to manipulate parallel computation plans. Msc thesis, Department of Computing, Imperial College London, U.K. (2001)
25. Lengauer, C.: Program optimization in the domain of high-performance parallelism (2004) In this volume.
26. Veldhuizen, T.L.: C++ templates as partial evaluation. In: PEPM '99: Partial Evaluation and Semantic-Based Program Manipulation. (1999) 13–18
27. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL Implementation in MetaOCaml, Template Haskell, and C++ (2004) In this volume.
28. Visser, E.: Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9 (2004) In this volume.